Developing Efficient Linked List Operations in the PETSc Library

Surtai Han
Hinsdale South High School

November 2012

Abstract

The software libraries that are used in large-scale scientific applications are often written for Unix-like operating systems, much to the dismay of beginner developers who often only have access to Microsoft Windows. The objective of this work is to develop efficient linked list operations in a library, called PETSc, using only a Windows PC. The first solution, a Linux virtual machine, was safe for the host operating system but heavily lacked in performance. On the other hand, the relatively risky dual-boot Linux installation had a performance superior to even that of the host machine. After the required software (such as text editors and the software library itself) was installed on the Linux OS, the development could begin. To improve upon the insertion time of the existing linked list in the PETSc library which used a linear search (O(n) IF statements), a binary search array was implemented. The new algorithm can search the list for the correct location using log(n) IF statements but takes O(n) iterations (non-IF) to update. In the end, the binary search approach was slightly less efficient than the linear search but more efficient than the other proposed linked list algorithms in the library. Elements that needed to be inserted often formed clusters where a linear search would be effective. Also, updating the binary search array was more expensive than anticipated because of the large jumps across distant memory locations.

Table of Contents

1. Introduction

Today, most large-scale scientific computing projects require high performance supercomputers that can easily reach trillions of calculations per second. Although improvements in hardware technologies have hugely contributed to the field of scientific computation, improvements in software are equally significant. For instance, parallel computing software makes it possible to boost performance without any improvement in hardware. The Portable, Extensible Toolkit for Scientific Computation[1] (PETSc) library is a software library that contains data structures and routines to help build and run these large projects. Sparse matrix operations are computational kernels of the PETSC library. Among these fundamental operations, symbolic matrix-matrix multiplications and factorization require efficient linked list insertion and retrieval. One of the objectives of this work is to improve the speed of the linked list operations for increased performance in everything that uses sparse matrix-matrix multiplication. It was hypothesized that implementing a binary search array (binary search algorithm) would increase the efficiency. Although the binary implementation would need an additional O(n) operations per array, it would require significantly less IF statements than the original linear search.

This paper also describes the various requirements have to be met in order to develop in the PETSc library on a Windows PC and includes installation instructions for the operating system itself, the PETSc library, and other software.

2. Prerequisites

There are some terms that need to be made clear before this paper can be effectively read.

Software refers to a collection of computer programs and related data that provides the computer with instructions for a specific task. Computer software is entirely different from computer **hardware**, which refers the physical devices (such as circuit boards, processors, memory) that make up a computer.

- **Emacs**[2] is a family of very extensible (expandable) text editors. The most popular version of Emacs is GNU Emacs which is the version used in this work.

- **Mercurial**[3] is a cross-platform, revision control tool (mainly implemented in Python) for software developers. Mercurial was used in this work to manage the local version of the PETSc library and to control the source code of the edited linked list.

- **Valgrind**[4] is a GPL licensed programming tool for memory debugging, memory leak detection, and profiling. It was installed to help detect flaws in the code that created memory leaks.

Computer Science refers to the design of software and algorithms. It is a different field from computer engineering (which focuses on the design of hardware) although the two sometimes overlap.

- **Parallel Computing** is a form of computation in which many calculations are carried out simultaneously. Large problems are divided into smaller ones, which are then solved concurrently, or, "in parallel".

Memory refers to the physical devices used to store electronic data. Computer memory is organized in a structure called a **memory hierarchy** where many levels of memory exist. The memory can be accessed increasingly faster towards the top of the pyramid, while there is an increasing amount of memory available towards the base.

The four major storage levels:

1. Internal – Processor registers and cache.
2. Main – the system RAM and controller cards (includes virtual memory)
3. Secondary storage such as hard drives.
4. Tertiary storage such as an external hard drive or cloud storage.

The term is used in this paper when discussing performance issues in algorithms related to lower level programming constructs involving locality of reference. Memory most often refers to a computer's Main memory.

- **Random Access Memory** (**RAM**) a form of computer data storage where storage and access times stay roughly constant regardless of the memory location. A device such as a magnetic tape that requires increasing time to access data stored on parts of the tape that are far from the ends would not be considered random access. However, more specifically, RAM mostly refers to **volatile** types of memory, where stored information is lost if the power is removed. In most personal computers, **DRAM** (dynamic) is preferred over **SRAM** (static) despite the power conservation and better performance that SRAM offers because of price. Double data rate synchronous dynamic random-access memory (**DDR SDRAM**), is a more complex and efficient form of DRAM that's widely used in PCs. Volatile random access memory is usually a computer's **main memory**.

- The **hard drive** (one or more rigid, rapidly rotating discs coated with magnetic material) serves mostly as secondary storage because it is comparatively slower and cheaper.

- **Virtual memory** refers to primary memory stored on secondary memory. In other words, it is a portion of the hard drive that is used as if it were RAM. It is considerably slower because not only is the I/O time of the hard drive slow, but there is also a constant shifting of memory from RAM to HDD. Virtual RAM allows more demanding applications to run, but with the cost of efficiency. Lack of RAM on the host computer often makes virtual machines slow and inefficient.

- A **cache** is a component that transparently stores data so that future requests for that data can be served faster. The data that is stored within a cache might be values that have been computed earlier or duplicates of original values that are stored elsewhere. If requested data is contained in the cache (cache hit), this request can be served by simply reading the cache, which is comparatively faster. Otherwise (cache miss), the data has to be recomputed or fetched from its original storage location, which is comparatively slower.

- A **processor register** is a very small amount of storage available in a CPU. Processor registers are normally at the top of the memory hierarchy, and provide the fastest way to access data. Almost all computers load data from a larger memory into registers where it is used by some machine instruction. Manipulated data is then stored back in main memory.

- A **CPU cache** is a cache used by the processor to increase memory access speed. The cache is a smaller, faster memory which stores copies of the data from the most frequently used main memory locations.

- **Locality of reference** is a common property of computer programs: The same values are often repeatedly accessed. Holding these frequently used values in caches improves performance, making fast registers and caches useful.

- The **Windows Registry** should not be confused with a processor register. It’s not a form of memory, but rather a database that stores configuration settings on Microsoft Windows operating systems.

- **Defragmenting** refers to the physical reorganization of the contents of a disk to store files in a fewer number of contiguous regions (fragments). It usually also creates larger regions of free space.

- **Disk partitioning** is dividing a hard disk drive into multiple storage units (partitions) in order to treat one physical disk drive as if it were multiple disks.

A **library** is a collection of resources such as subroutines and values used by programs on a computer, often to develop software.

- **Message Passing Interface** (**MPI**)[5] is a library specification for portable message-passing systems designed to function on a wide variety of parallel computers that are used to run large-scale applications. Several implementations of MPI are free and in the public domain.

- **Basic Linear Algebra Subprograms** (**BLAS**)[6] is an application programming interface standard for publishing libraries to perform basic linear algebra operations such as matrix multiplication. These subprograms are often used in high-performance computing and are used to build larger packages such as LAPACK.

- **Linear Algebra Package** (**LAPACK**)[7] is a library for numerical linear algebra. The routines handle both real and complex matrices in both single and double precision.

- **PETSc** (**Portable, Extensible Toolkit for Scientific Computation**)[1] is a suite of data structures and routines for the scalable (parallel) solution of scientific applications. It is built on top of MPI, BLAS, and LAPACK. PETSc is intended for use in large-scale application projects. It includes a large collection of parallel linear and nonlinear equation solvers that are easily used in application codes written in C, C++, Fortran and Python. PETSc also provides many of the mechanisms needed within parallel application code.

A **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently.

- A **linked list** is a data structure consisting of a group of nodes where each node is composed of a datum and a link to the next node in the sequence.

- An **array** is a data structure consisting of a collection of elements that uses physical memory addresses consecutively. Each element is identified by an array index.

- A **gap buffer** is a dynamic array that allows efficient insertion and deletion operations clustered near the same location. Gap buffers are especially common in text editors, where most changes to the text occur at or near the current location of the cursor. The text is stored in a large buffer in two contiguous segments, with a gap between them for inserting new text. Emacs uses this data structure to edit text. “Save current buffer to its file” simply means “save changes”.

An **operating system** is a set of software that communicates with the computer hardware and provides common services for computer programs. For example, Microsoft Windows is an operating system while Microsoft Word is an application.

- A **kernel** is the main component of most computer operating systems; it is a bridge between applications and the actual data processing done at the hardware level. An operating system includes software other than the kernel. For example, Ubuntu, Redhat, and Debian are all operating systems that use the Linux kernel.

- **Booting** is the initial set of operations that a computer system performs when electrical power is switched on. Part of booting requires finding, loading and starting an operating system which is supplied by the Ubuntu CD in this case.

- **Bit** means 0 or 1. 32-bit or 64-bit is referring to the size of address buses, data buses, integer size, and registers. A 32-bit operating system can use up to 4 gigabytes of RAM while a 64-bit machine can access potentially 17.2 billion gigabytes. With 32 bits, there are only 2^32 many possibilities for memory addresses. For example, 10101011010010000101001110010011 is one possibility in 32-bit processors while 1010101101001000010100111001001110101011010010000101001110010011 would be one possibility in 64-bit processors. The bit of an operating system indicates what processor it is compatible with. 32 Bit Operating systems work on 32 Bit CPUs and 64 Bit CPUs, but 64 Bit operating systems work on 64Bit CPUs only.

- **Linux**[8] here is referring to any operating system with a Linux kernel. There is an operating system that is called Linux itself but Ubuntu was the guest OS used in this project. Operating systems that use the Linux kernel are Unix-like and assembled under the model of free and open source software development and distribution. Unix-like

operating systems are widely used in research because of their flexibility, price, and stability.

- **Ubuntu**[9] is a popular computer operating system that uses the Linux kernel and is distributed as free and open source software using its own desktop environment.

- **Grub**[10] is the boot loader of the GNU Project. In this work, it is used to choose which operating system to boot from once two operating systems are installed onto the same hard drive.

- **Dual Boot** is the act of installing two operating systems onto the same hard drive. Often, a custom boot loader is needed.

A **shell** is software that provides an interface for users to access the services of a kernel.

- **Bash**[11] is a Unix shell written for the GNU Project. It is distributed widely as the default shell on Linux. Bash is a command processor, typically run in a text window, that executes the commands the user types. It "bashed" together the features of the sh, csh and ksh shells.

A **virtual machine** is a completely isolated guest operating system installation within a normal host operating system. In other words, it's a software implementation of a machine that executes programs like a physical machine.

- **VirtualBox**[12] is a free software that allows a computer to create and run multiple virtual machines. Oracle VM VirtualBox is installed on an existing host operating system as an application; this host application allows additional guest operating systems to be loaded and run, each with its own virtual environment.

- An **ISO** (.iso) file is an archive file (also known as a disk image) of an optical disc, composed of the data contents of every written sector of an optical disc including the optical disc file system. ISO images can be created from optical discs and those images can be used to write to other optical discs. In this project, an iso file of an operating system is burned onto a disc to boot from.

The **GNU Project**[13] is a free software, mass collaboration project. The founding goal of the project was to develop "a sufficient body of free software [...] to get along without any software

that is not free." The Linux kernel that was released as free software completed GNU's goal of a free software operating system.

- The **GNU General Public License**[14] (GNU GPL or simply GPL) is the most widely used free software license.

A **programming language** is an artificial language designed to communicate instructions to a machine, particularly a computer.

- A **high-level programming language** is a programming language with strong abstraction from the details of the computer. The syntax of high-level programming languages is similar to everyday speech. Most popular programming languages today, such as Java, C++, and Python, are considered high-level programming languages.

- A **low-level programming language** is a programming language that provides little or no abstraction from a computer's instruction set architecture. Generally this refers to either machine code or assembly language. Low-level languages are sometimes described as being "close to the hardware."

- A **compiler** is software that translates high-level, source code written in a programming language into a lower-level language such as assembly or machine code (computer code) that the machine can execute.

- **Pseudocode** is an informal high-level description of a computer algorithm. It is intended for easier human reading and understanding of the key principles of the algorithm, but retains the structural conventions of a programming language. It is used in this paper to roughly explain how the newly proposed algorithms work.

- **C** is a general-purpose, procedural programming language. Because its design provides constructs that map efficiently to typical machine instructions, it found lasting use in applications that had formerly been coded in assembly language. At the same time, C is also capable of abstraction like high-level languages. Thus, C is often considered the "middle level" language. C is also one of the most widely used programming languages of all time-both the Unix and Linux operating systems are written in C as well as a large portion of the PETSc library.

- **C++** is another general-purpose programming language. It is also regarded as an intermediate-level language, because it retained the high-level and low-level features of

C. C++ adds object oriented features and other enhancements to the C programming language, hence the name C++.

- **Fortran**[15] is a general-purpose programming language that is especially suited to numeric computation and scientific computing.

- **Python**[16] is another general-purpose, high-level programming language that is used in mercurial and PETSc.

3. Software Installation

Installing Linux Using a Virtual Machine

- Download and install virtual machine (VM) software such as VirtualBox
- Download the latest version of Ubuntu as an ISO file. Make sure to download the 32-bit version even if the windows host operating system is 64-bit because the 64-bit version of Ubuntu is often incompatible with certain software
- Burn the Ubuntu ISO file onto an unused CD
- Create a new virtual machine on VirtualBox and run it. Boot from the cd that was created in order to install Ubuntu onto the newly created virtual machine (internet connection required and several gigabytes of hard disk space recommended)
- Note that the virtual machine implementation of Ubuntu is significantly slower than the dual boot.

Installing Linux Using a Disk Partition

- Download the latest version of Ubuntu as an ISO file. Make sure to download the 32-bit version even if the windows host operating system is 64-bit
- Burn the Ubuntu ISO file onto an unused CD and eject the CD
- Because the following procedure is risky, it is recommended to back up all important files in an external hard drive or another medium
- Defragment the hard drive
- Shrink the hard drive partition using Windows (recommended) or third-party software such as GParted
- Shut down the system immediately after partitioning
- Boot from the Ubuntu CD and follow the instructions to install Ubuntu onto the unused partition

Installing PETSc and other software utilities required for development on Ubuntu

- Install C/C++ and Python compilers using “sudo apt-get install build-essential” on the terminal
- Install Emacs using “sudo apt-get install emacs” (text editor)
- Install Mercurial using “sudo apt-get install mercurial” (source control software)
- Install Valgrind using “sudo apt-get install Valgrind” (memory debugging)
- Install CMake using “sudo apt-get install cmake” (builds libraries faster)
- Next, edit the file “.bashrc” located in the home directory. Add “export PETSC_DIR=$HOME/soft/petsc-dev” (where the local copy of the petsc library will be located) and “export PETSC_ARCH=arch-linux-debug” (name of the folder in the petsc directory where the code will be located) Theses two lines of code will initialize the two variables whenever a new terminal is opened.
- Restart the terminal
- It is important to download PETSc into a folder other than the home directory.
- Navigate to the directory where PETSc is going to be installed
- hg clone http://petsc.cs.iit.edu/petsc/petsc-dev
- cd petsc-dev
- nano .hg/hgrc (and add the following lines)

  [hooks]
  post-pull = "$HG" $HG_ARGS -u $@ --cwd config/BuildSystem

- hg clone http://petsc.cs.iit.edu/petsc/BuildSystem config/BuildSystem

- To configure, use “./configure --with-cc=gcc --with-fc=0 --download-f2cblaslapack --download-mpich --with-c2html=0 –with-cmake=/path/to/cmake”
- The Fortran compiler and html documentation functions will not be needed
- Afterwards, use “make all test” to compile and test the library
- With the current settings, debugging is turned on. To collect accurate performance data, another library with debugging turned off is needed. To do this, set PETSC_ARCH equal to “arch-linux-opt” (folder where the optimized version of PETSc will be installed) and configure the library again with the same settings but at the end, add “--with-debugging=0”
- Again, use “make all test” to compile and test the library

4. Algorithm

The goal is to create a linked list data structure in the PETSc library for storing non-negative integers. The list cannot allow duplicates (many will be present in the input array) and should

remain sorted at all times. The focus of the algorithm is to efficiently merge sorted arrays into the list.

The existing linked list data structure in PETSc is stored in a pre-allocated array so that new memory does not have to be allocated each time a new value is added. A separate bit array assists in keeping track of the values that already exist in the list. To insert elements, the algorithm first checks the bit array to see if the value already exists. If not, a linear search is performed on the linked list to find the correct insertion location. The new value is appended to the end of the data array and the pointers are adjusted.

To increase search speed for the insertion location, a sub-array of indexes (memory locations) of the elements in ascending order is added on to the end of the data structure. This new array, named the binary search array (**bsearch**), is searched using the binary search algorithm and updated after each array is merged.

To differentiate from the existing linked list in the library (**LList**), the linked list that uses the binary search algorithm is named **LList_bsearch**.

How the Linked List is stored in an array

The data structure uses a single array instead of parallel arrays to minimize jumping back and forth distant memory locations. A few critical values are held at the start of the array for efficient access. The first index (array[0]) contains the number of nodes currently stored in the list excluding the head node. The second value (array[1]) contains the location in the array where the binary search list starts, a special “sub-array” used to perform a binary search on the list when a new element needs to be added. The next two values represent the head node: array[2] contains the max integer the list can contain while array[3] contains a pointer back to itself. array[4] contains the data of the first real node.

In Figure 1, each blue oval represents a node. The number on the left is the data that the node contains while the value on the right is the memory location of the next node in ascending order. There is space for just one more element. The binary search array (orange) is stored at the end. The first value in the binary search section represents the memory location of the smallest node. The next value represents the memory location of the node immediately greater than the first.

The bitarray is a separate data structure (not shown in diagrams) that needs to be passed in separately to a method whenever the linked list is used. The length of the bit array is the maximum value that the list can contain. An example usage of the bitarray would be:
If bitarray[9] returns 0, that means the integer “9” is not present in the linked list.

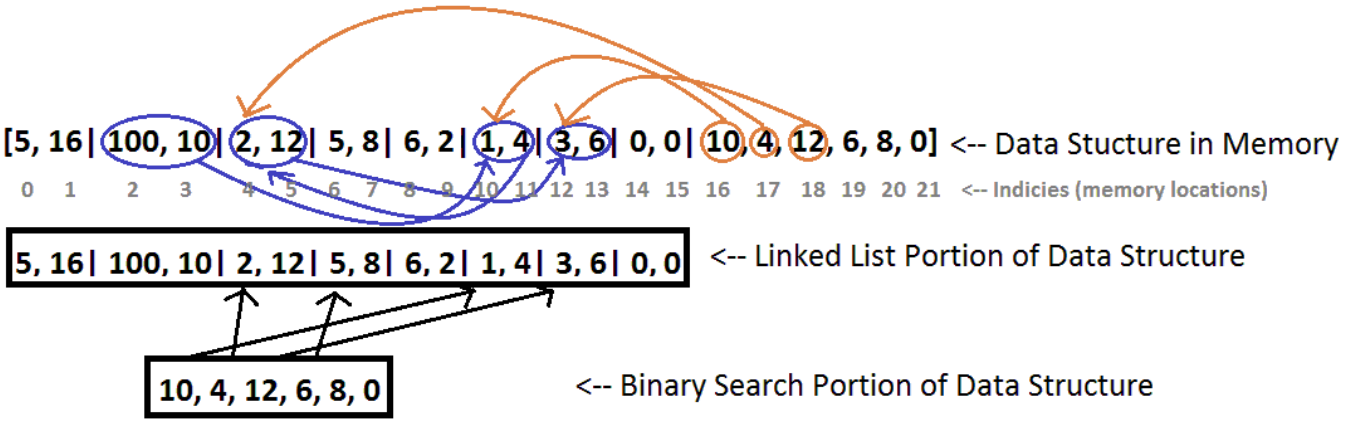

Figure 1

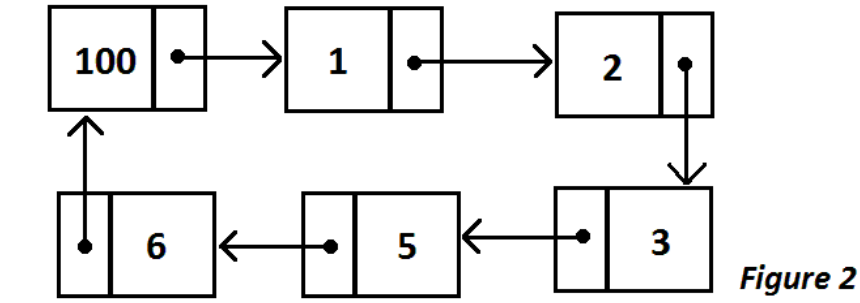

Figure 2

In Figure 2, an abstract representation of the linked list is shown. Each rectangle represents a node with a datum and a pointer, or address to the next node.

Figure 3 depicts the binary search array. It is an array of pointers that point to the nodes in ascending order. Thus, a binary search can be carried out on the binary search array and the index of the correct insertion location can be quickly found.

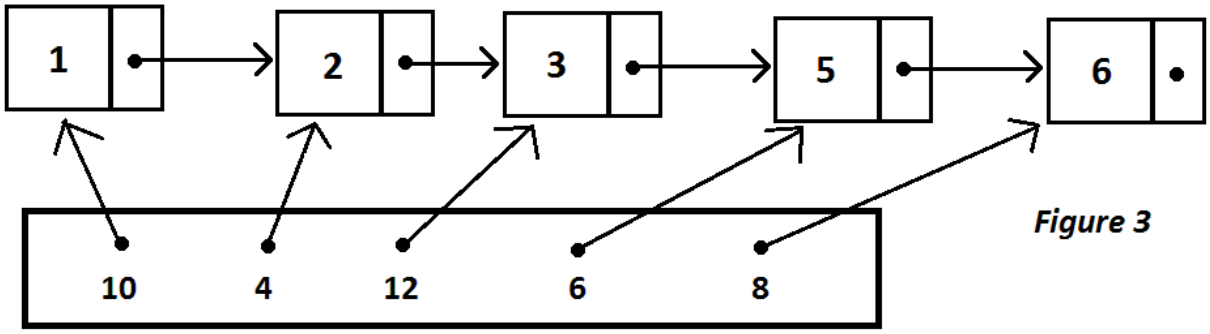

Figure 3

Insertion of new element (given insertion location)

This algorithm (used in both the linear search and binary search algorithms) appends the new value to the end of the data array and adjusts the pointers to insert the element into the linked list. The memory location of the small node pointer, the location of the big node data, and the number of nodes already stored in the list are required in order for a new element to be inserted.

Algorithm 1: Insertion {

1) let the small node point to the memory location in the array where the new value will be inserted
2) append the new value to the array by (put it in the newly calculated memory location)
3) let the new node point to the node that is immediately bigger than itself

}

In Figure 4, 7 is the new value, 6 is the number that is immediately less than the new value (the small node), and 12 is the value contained in the big node (the one that is immediately greater than the new value).

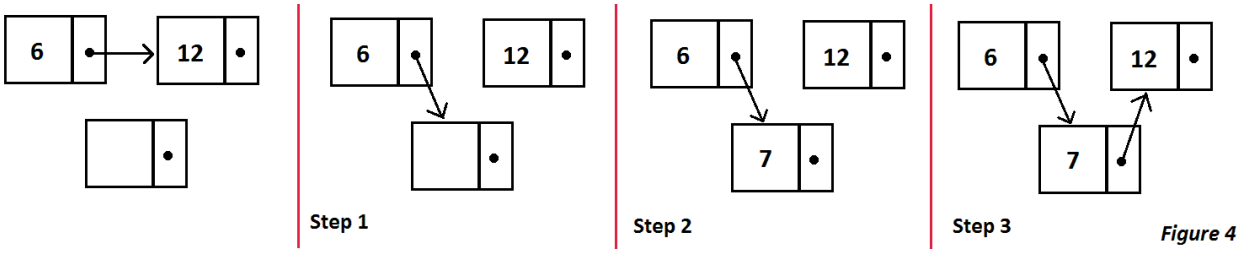

Figure 4

Basic Linear Search

This algorithm iterates through the nodes of the linked list one by one to find the location that the new element should be added to and inserts the new element.

Algorithm 2: Basic Linear Search {

Start the algorithm at the head node
Store the number of entries that are in the linked list in a local variable
Iterate through input array
{

```
                Get the next value in the input array
                if the input value does not already exist in the linked list
                {
                        record that the new value has been added
                        do {
                                record where the pointer of small node is located
                                move to the next node
                                get the value in the next node
                        } while (new value > value of current node)
                        insert new value using Algorithm 1: Insertion
                }
        }
        update the number of entries in the linked list according to local value
}
```

In Figure 5, the linked list is represented by nodes and the binary search array is not shown because is it not part of the basic linear search. The red squares highlight the nodes that the algorithm is currently on. The blue box highlights the new value (from the input array) that needs to be inserted into the list.

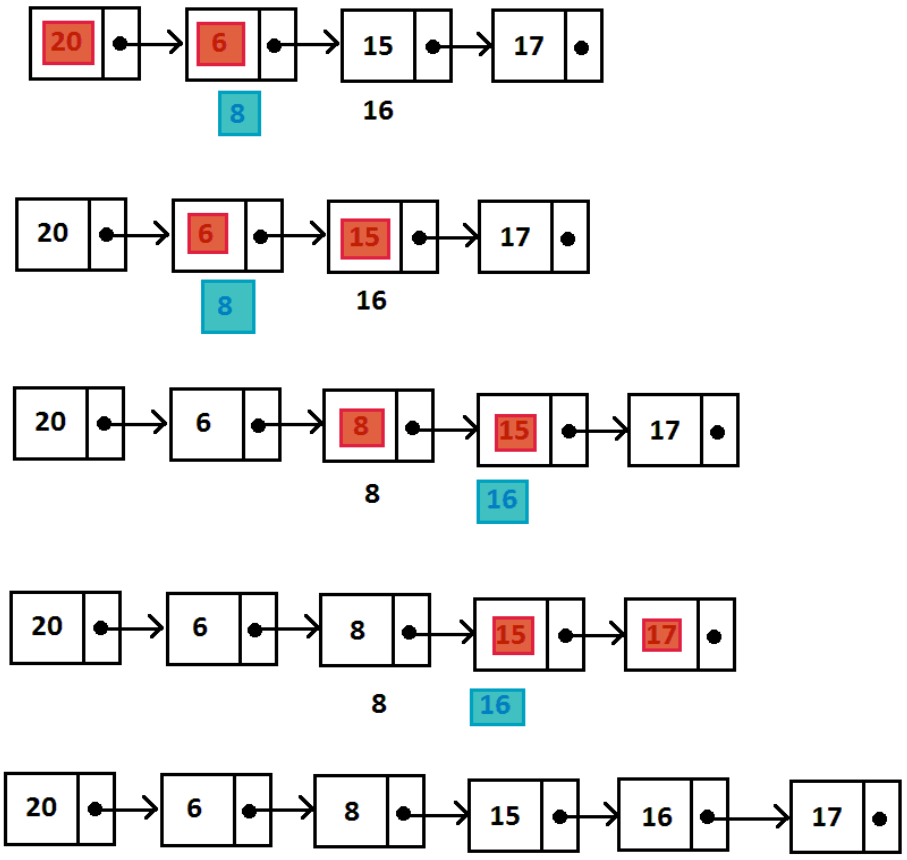

Figure 5

Retrieving data and Cleaning the list

The purpose of the linked list was is to add items quickly. Now, all of the items need to be retrieved as an array and the list has to be reinitialized so that it can be used again. It should be noted that resetting does not imply setting all the values to zero. An adjustment of just a few values is sufficient. For example, set the number of entries to 0.

Binary Search

This algorithm performs the binary search algorithm on the binary search array to find the correct location for the new element. The algorithm works by dividing the array size in half to find the middle value. If the new value is smaller, the bigger half is discarded. Otherwise, the smaller half is discarded. The section that is kept is then divided in two and searched again until the search is narrowed down to two nodes. A linear search then starts from the smaller of the two nodes.

Algorithm 3: Binary Search {

```
    While ( remaining array is size 2 or more )
    {
        if new value is smaller than the middle value
            discard the upper portion of the array
        else
            discard the lower portion of the array
    }
```

    Algorithm 2: Linear Search starting from the lesser value in the remaining array
}

Figure 6 depicts the insertion of the value “35” in an array that contains values 0-100. First, the greater half is discarded because 35 is less than 50. Then the lesser half is discarded because 35 is greater than 25. And then the greater half is discarded again because 35 is less than 37. Eventually, only two values will remain.

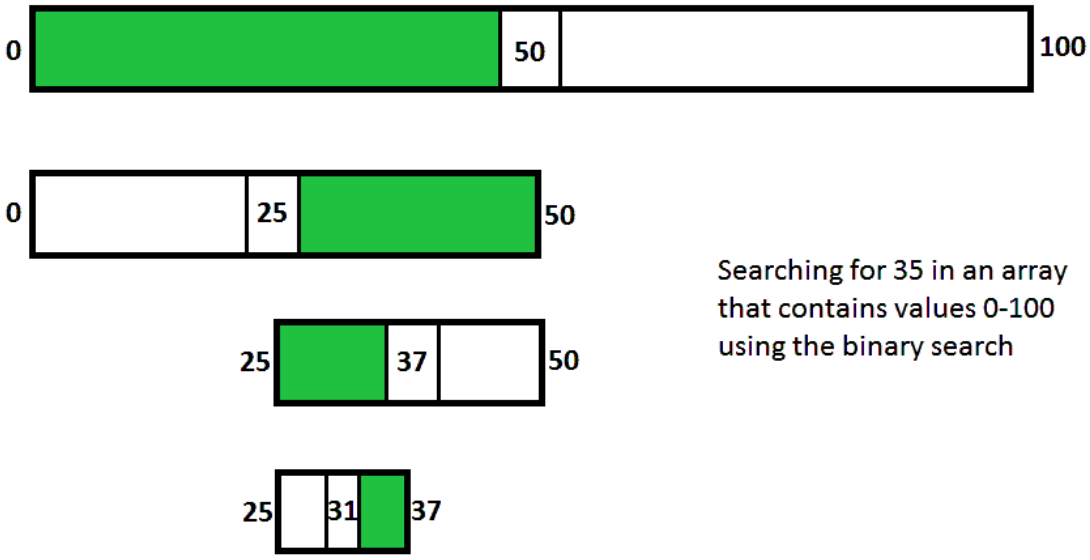

Figure 6

Updating the binary search array

After an input array is merged into the list, the binary search array should be updated for good performance in merging the successive input array.

Algorithm 4: Bsearch Update {

```
        Iterate (size of linked list) number of times
        {
                Get the location where the next value in the linked list exists
                Record the location in the binary search array
        }
}
```

If the binary search array were not updated, the algorithm would still work properly but it would not be as efficient because a new element might need to be inserted in a location where there is a cluster of elements that are not known to the binary search array.

In Figure 7, the yellow nodes are the ones that are unknown to the binary search array yet exist in the linked list. If the number “106” needed to be added, the binary search would stop at “89” and there would have to be two more additional steps in the linear search before the insertion location for the “106” is found.

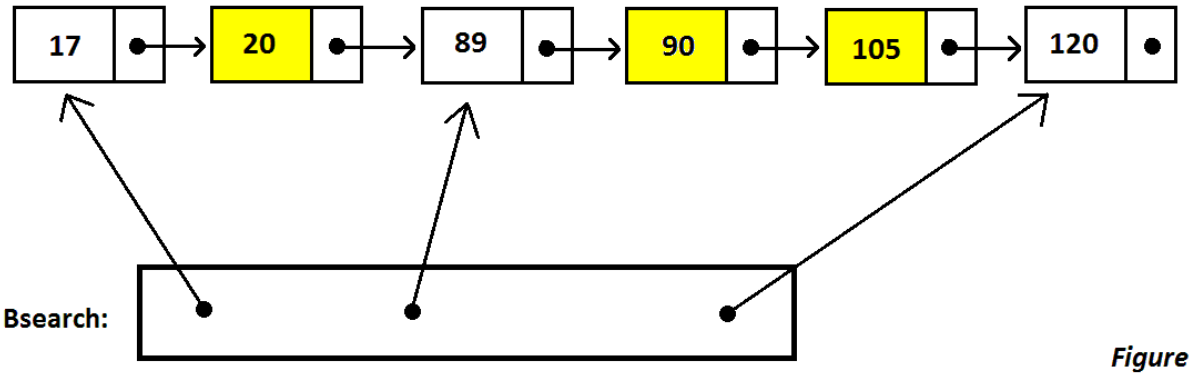

Figure 7

5. Numerical Experiments and Results

To achieve optimal performance and collect the most accurate data on actual efficiency, optimized mode (o-mode) needs to be used instead of debugging mode (g-mode) when building the PETSc library and application codes. Code compiled under these settings is more difficult to debug but boasts significant performance improvement.

The experiments were conducted using the matrix "arco4" of size 27,007x 27,007 from PETSc matrix collection. The matrix arises from multiphase flow modeling of oil reservoirs. It is sparse with 543103 non-zero elements, i.e. only 543103/(27007x27007) = .07% non-zero matrix elements. The table below shows the numerical results collected from a Windows Vista machine with an Intel Duo Core processor.

Search Algorithm	Execution Time (sec) (debugging mode)	Execution Time (sec) (optimized mode)
Linear search	6.8e-01	2.28e-01
Binary search	1.04 e+00	3.32 e-01
Bit heap	1.47e+00	5.24e-01
Heap	7.44e+00	1.96e+00

The proposed binary search approach was slightly less efficient than the linear search but more efficient than the other existing linked list algorithms in the library. Elements that needed to be inserted often formed clusters where a linear search would be effective. Also, updating the binary search array was more expensive than anticipated because of the large jumps across distant memory locations.

6. Conclusion and Future Work

Large organizations and scientific computation alike prefer Unix over Windows because Unix is portable, stable, cheap (often free), open-source, and possess much greater processing power. For developers looking to run a Unix-like operating system, a dual boot installation, if done correctly, will be a much better long term solution.

The issue with the binary search algorithm was that it was not specific enough to the problem. It was perhaps better suited for merging unsorted arrays where there is potentially much space between one insertion location and the next. A second approach to this problem better accounted for the fact that, for the most part, there are few nodes between one insertion location and the next. This approach involved a similar linear search but iterated through the linked list two nodes at a time instead of one. The performance was still 1.1x slower than the original linear search but it was faster than the binary, which was around 1.4x slower. We plan to continue this research on efficient methods of linked list insertion to increase the efficiency of not only the linked list algorithm itself, but also various other subroutines that heavily rely on fast linked list performance.

7. Acknowledgements

A huge thanks to my instructor, Professor Hong Zhang of Computer Science Department in Illinois Institute of Technology and Mathematics and Computer Science Division at Argonne National Laboratory.

8. References

[1] http://www.mcs.anl.gov/petsc/
[2] http://www.gnu.org/software/emacs/
[3] http://mercurial.selenic.com/
[4] http://valgrind.org/
[5] http://www.mcs.anl.gov/research/projects/mpi/
[6] http://www.netlib.org/blas/
[7] http://www.netlib.org/lapack/

[8] http://www.linuxfoundation.org/
[9] http://www.ubuntu.com/
[10] http://www.gnu.org/software/grub/
[11] http://www.gnu.org/software/bash/manual/bashref.html
[12] https://www.virtualbox.org/
[13] http://www.gnu.org/gnu/thegnuproject.html
[14] http://www.gnu.org/licenses/gpl.html
[15] http://www.fortran.com/
[16] http://www.python.org/

9. Appendix

```
#undef __FUNCT__
#define __FUNCT__ "PetscLLCondensedCreate_binarysearch"
/*
  Create and initialize a condensed linked list -
    same as PetscLLCreate(), but uses a scalable array 'lnk' with size of max number of entries, not O(N).
    Contributed by Surtai Han, Hinsdale High School

  Input Parameters:
    nlnk_max  - max length of the list
    lnk_max   - max value of the entries
  Output Parameters:
    lnk       - list created and initialized
    bt        - PetscBT (bitarray) with all bits set to false. Note: bt has size lnk_max, not nln_max!
*/
PETSC_STATIC_INLINE PetscErrorCode PetscLLCondensedCreate_binarysearch(PetscInt
nlnk_max,PetscInt lnk_max,PetscInt **lnk,PetscBT *bt)
{
  PetscErrorCode ierr;
  PetscInt       *llnk;
  PetscFunctionBegin;
  ierr = PetscMalloc(2*(nlnk_max+2)*sizeof(PetscInt)+(nlnk_max)*sizeof(PetscInt),lnk);CHKERRQ(ierr);
  ierr = PetscBTCreate(lnk_max,bt);CHKERRQ(ierr);
  llnk = *lnk;
  llnk[0] = 0;        /* number of entries on the list */
  llnk[1] = 2*(nlnk_max+2); /*location where the binary search array begins*/
  llnk[2] = lnk_max;   /* value in the head node */
  llnk[3] = 2;        /* next for the head node */
  llnk[llnk[1]] = 2;   /* first value of bsearch array is 2 */
```

```
  PetscFunctionReturn(0);/*bsearch array is nlnk_max long*/
}

#undef __FUNCT__
#define __FUNCT__ "PetscLLCondensedAddSorted_binarysearch"
/*
  Add a SORTED ascending index set into a sorted linked list. See PetscLLCondensedCreate() for detailed description.
  Input Parameters:
    nidx      - number of input indices
    indices   - sorted interger array
    lnk       - condensed linked list(an integer array) that is created
    bt        - PetscBT (bitarray), bt[idx]=true marks idx is in lnk
  output Parameters:
    lnk       - the sorted(increasing order) linked list containing previous and newly added non-redundate indices
    bt        - updated PetscBT (bitarray)
*/
PETSC_STATIC_INLINE PetscErrorCode PetscLLCondensedAddSorted_binarysearch(PetscInt nidx,const PetscInt indices[],PetscInt lnk[],PetscBT bt)
{
  PetscInt _k,_entry,_location,_next,_lnkdata,_nlnk,_newnode,_min,_max,_mid;
  PetscFunctionBegin;
  _nlnk     = lnk[0]; /* num of entries on the input lnk */
  _min      = lnk[1]; /* where the binary search array starts */
  _max      = lnk[0]+lnk[1]; /* where the binary search array ends */
    for (_k=0; _k<nidx; _k++){
      _entry = indices[_k];
      if (!PetscBTLookupSet(bt,_entry)){  /* new entry */
        /* search for insertion location using binary search*/
          if(_entry>lnk[lnk[3]]){ /* if bigger than first value */
            _max=lnk[1]+lnk[0];
            while (_max > _min+1){
              _mid=(_max+_min)/2;
              if(_entry > lnk[lnk[_mid]])/*entry greater than midpoint*/
                _min=_mid;
              else/*entry less than midpoint*/
                _max=_mid-1;
            }
            _location=lnk[_min];/*start linear search from min*/
          }
          else{
```

```
        _location=2;/*otherwise, start lin search from head node*/
       }
      do{
       _next=_location+1;
       _location=lnk[_next];
       _lnkdata=lnk[_location];
      }while(_entry > _lnkdata);
      /* insertion location is found, add entry into lnk */
      _newnode       = 2*(_nlnk+2);   /* index for this new node */
      lnk[_next]     = _newnode;     /* connect previous node to the new node */
      lnk[_newnode]   = _entry;       /* set value of the new node */
      lnk[_newnode+1] = _location;    /* connect new node to next node */
      _nlnk++;
    }  \
   }\
  lnk[0]   = _nlnk;   /* update number of entries in the list */

  _location=lnk[3]; /*location of data of first node */
  _min=lnk[1];/*location where the binary search array starts */
  for(_k=0; _k<_nlnk; _k++){
   lnk[_k+_min]=_location;
   _location=lnk[_location+1];/*update bsearch array*/
  }
  PetscFunctionReturn(0);
}

#undef __FUNCT__
#define __FUNCT__ "PetscLLCondensedClean_binarysearch"
PETSC_STATIC_INLINE PetscErrorCode PetscLLCondensedClean_binarysearch(PetscInt lnk_max,PetscInt
nidx,PetscInt *indices,PetscInt lnk[],PetscBT bt)
{
  PetscErrorCode ierr;
  PetscInt       _k,_next,_nlnk;
  PetscFunctionBegin;
  _next = lnk[3];      /* head node */
  _nlnk = lnk[0];      /* num of entries on the list */
  for (_k=0; _k<_nlnk; _k++){
   indices[_k] = lnk[_next];
   _next       = lnk[_next + 1];
   ierr = PetscBTClear(bt,indices[_k]);CHKERRQ(ierr);
  }
  lnk[0] = 0;         /* num of entries on the list */
```

```
  lnk[2] = lnk_max;    /* initialize head node */
  lnk[3] = 2;          /* head node */
  lnk[lnk[1]] = 2;   /* first value of bsearch array is 2 */
  PetscFunctionReturn(0);
}

#undef __FUNCT__
#define __FUNCT__ "PetscLLCondensedView_binarysearch"
PETSC_STATIC_INLINE PetscErrorCode PetscLLCondensedView_binarysearch(PetscInt *lnk)
{
  PetscErrorCode ierr;
  PetscInt       k;

  PetscFunctionBegin;
  ierr = PetscPrintf(PETSC_COMM_SELF,"LLCondensed of size %d, (val,  next)\n",lnk[0]);CHKERRQ(ierr);
  for (k=2; k< lnk[0]+2; k++){
    ierr = PetscPrintf(PETSC_COMM_SELF," %D: (%D, %D)\n",2*k,lnk[2*k],lnk[2*k+1]);CHKERRQ(ierr);
  }
  PetscFunctionReturn(0);
}

#undef __FUNCT__
#define __FUNCT__ "PetscLLCondensedDestroy_binarysearch"
/*
  Free memories used by the list
*/
PETSC_STATIC_INLINE PetscErrorCode PetscLLCondensedDestroy_binarysearch(PetscInt *lnk,PetscBT bt)
{
  PetscErrorCode ierr;
  PetscFunctionBegin;
  ierr = PetscFree(lnk);CHKERRQ(ierr);
  ierr = PetscBTDestroy(&bt);CHKERRQ(ierr);
  PetscFunctionReturn(0);
}
```